Technical Report 990

Computerized Testing System Software Conversion and Development: Identifying Software and Hardware Portability Issues and Solutions

Rodney Rosse
American Institutes for Research

David Dodd
STATCOM, Inc.

Jay M. Silva
U.S. Army Research Institute

October 1993

United States Army Research Institute for the Behavioral and Social Sciences

U.S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency Under the Jurisdiction of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Director

Research accomplished under contract
for the Department of the Army

STATCOM, Inc.

Technical review by

Peter J. Legree
Alan F. Drisko

NOTICES

DISTRIBUTION: Primary distribution of this report has been made by ARI. Please address correspondence concerning distribution of reports to: U.S. Army Research Institute for the Behavioral and Social Sciences, ATTN: PERI-POX, 5001 Eisenhower Ave., Alexandria, Virginia 22333-5600.

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U.S. Army Research Institute for the Behavioral and Social Sciences.

NOTE: The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1993, October	Interim Jul 91 - Dec 92

4. TITLE AND SUBTITLE
Computerized Testing System Software Conversion and Development: Identifying Software and Hardware Portability Issues and Solutions

5. FUNDING NUMBERS
MDA903-91-D-0024
62785A
791
1211
A86

6. AUTHOR(S)
Rosse, Rodney (American Institutes for Research); Dodd, David (STATCOM, Inc.); and Silva, Jay M. (ARI)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
STATCOM, Inc.
7921 Jones Branch Road
Suite 445
McLean, VA 22102

8. PERFORMING ORGANIZATION REPORT NUMBER
--

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Institute for the Behavioral and Social Sciences
ATTN: PERI-RS
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARI Technical Report 990

11. SUPPLEMENTARY NOTES
Contracting Officer's Representative, Jay M. Silva.
American Institutes for Research, 3333 K Street, NW, Suite 300, Washington, DC 20007, is a subcontractor for STATCOM, Inc.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
--

13. ABSTRACT *(Maximum 200 words)*

This report identifies hardware and software issues associated with the transition of computerized tests from old hardware and software configurations to currently available hardware and software. Hardware issues addressed include real-time clock performance, storage devices, video adapter, central processing unit (CPU), and CPU processing speed. Software issues addressed include conversion from PASCAL to C code and hardware-dependent routines.

14. SUBJECT TERMS
Software conversion
Computerized test conversion
Portability
Computerized testing

15. NUMBER OF PAGES
64

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Technical Report 990

Computerized Testing System Software Conversion and Development: Identifying Software and Hardware Portability Issues and Solutions

Rodney Rosse
American Institutes for Research

David Dodd
STATCOM, Inc.

Jay M. Silva
U.S. Army Research Institute

Selection and Classification Technical Area
Michael G. Rumsey, Chief

Manpower and Personnel Research Division
Zita M. Simutis, Director

U.S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue, Alexandria, Virginia 22333-5600

Office, Deputy Chief of Staff for Personnel
Department of the Army

October 1993

Army Project Number
2Q162785A791

Manpower, Personnel, and Training

FOREWORD

Under Project A, a large-scale Army selection and classification project, various new constructs were identified, measured, and validated. Some of these measures were implemented on early 1980s computer hardware, programming language compilers, and operating systems. While these measures have proved useful in predicting various facets of performance, the hardware on which computerized tests were implemented is outdated and in disrepair. In addition, the original programming for these tests evolved through an iterative process to take advantage of the latest breaking technology. This resulted in software that benefited from available technology but was poorly documented.

Given this state of affairs, it was necessary to update the original experimental computerized testing system to address these and other issues. The goals of this effort are as follows: (1) convert the original program code to the current programming language standard, (2) extensively document the new code, (3) take advantage of new disk, processor, and video technologies, (4) code in a manner that isolates hardware-dependent code for future upgradability, (5) program only original computerized tests that, from validation studies, indicate the greatest usefulness for the Army, and (6) program an additional two promising spatial tests that were administered in paper-and-pencil format under Project A.

This report was generated to identify the hardware and software issues involved in porting the original software to new hardware and a new programming language compiler. Options were outlined for each issue and recommendations were made to effectively address them. These recommendations, if followed, will lead to the creation of a new experimental computerized testing system that will be useful today and easily upgraded in the future.

EDGAR M. JOHNSON
Director

ACKNOWLEDGMENTS

The authors express their appreciation to the following individuals who contributed to this project: From the U.S. Army Research Institute for the Behavioral and Social Sciences, Clinton B. Walker and Sidney A. Sachs, who provided critical historical guidance for the project, and from Rohr Applied Systems, Jay Rohr for his expertise, professionalism, and willingness to help.

COMPUTERIZED TESTING SYSTEM SOFTWARE CONVERSION AND DEVELOPMENT: IDENTIFYING SOFTWARE AND HARDWARE PORTABILITY ISSUES AND SOLUTIONS

EXECUTIVE SUMMARY

Requirement:

In the early 1980s, as part of a long-term selection and classification project named Project A, a battery of computerized tests was developed. Today these experimental computerized tests are still being researched and some may be added in future revisions to the Armed Services Vocational Aptitude Battery. However, the hardware for which these tests were developed has become obsolete, and a large proportion of the hardware is now in need of repair or is highly unreliable.

To continue using these tests, it is necessary to port the program code for them to an updated hardware standard. Because of the massive change in the personal computer in the last 10 years, changes in everything from disk, video, and processor need to be addressed. The new program code should also address the following areas: (1) conversion of the original program code to the current programming language standard, (2) documentation for the new code, and (3) isolation of hardware-dependent code for future upgradability.

In addition, this effort will convert only original computerized tests that, from validation studies, indicate the greatest usefulness for the Army. But it will also program an additional two promising spatial tests that were administered in paper-and-pencil format under Project A.

Procedure:

This effort will identify hardware and software portability issues associated with the transition of these tests from the original software and hardware configuration (early 1980s technology) to a current 386-486/MS-DOS hardware/software environment. Hardware issues include changes in the response pedestal, processor, and video. Software issues include items such as the compiler, operating system, and programming language, as well as advances in design and documentation.

Findings:

Many porting areas were addressed and solutions recommended. A summary of the recommendations is as follows:

1. the system will use IEEE 488.2 interface card and driver software to allow the use of the new RGI response pedestal. (In addition, the system will continue to accept the original Project A response pedestal),

2. instead of using an add-on timer chip at an added cost (as was done in the original system), the new system will achieve highly accurate timing by reprogramming the system timer to achieve millisecond timing resolution,

3. for localized security concerns, the system will be flexible and allow reading and writing of data to and from a variety of storage locations,

4. for future upgradability, the system will use video primitives to compartmentalize code dependent on the video adapter,

5. for processor speed independence, programs will not be affected by processor speed and will perform identically on processors running at different rates,

6. port will use a conceptual conversion of the software because of the lack of original documentation and to allow a reorganization of the code,

7. port will compartmentalize and create libraries for hardware-dependent code for such devices as the response pedestal and timer,

8. as in the original code, the new system will incorporate testing and calibration of hardware devices,

9. the new software will be documented extensively with in-line comments and software user's and programmer's manuals, and

10. to ensure functional specifications equal to that of the original computerized testing system, the original code and its performance will be examined.

Utilization of Findings:

The porting issues addressed were intended to be comprehensive. The porting effort will follow these recommendations to the greatest extent possible. Unanticipated issues or conflicts, if they arise, will be addressed by applying the general philosophy underlying the solutions to the identified design considerations.

COMPUTERIZED TESTING SYSTEM SOFTWARE CONVERSION AND DEVELOPMENT: IDENTIFYING SOFTWARE AND HARDWARE PORTABILITY ISSUES AND SOLUTIONS

CONTENTS

	Page
INTRODUCTION	1
Purpose of the Research	1
Organization of the Portability Project	1
Background	2
IDENTIFICATION OF HARDWARE PORTABILITY ISSUES (TASK I)	5
Current Hardware Configuration	5
Target Hardware Configurations	6
Issues of Portability Between the Current and Target Configurations	7
Issues of Portability to Other Hardware Configurations	9
IDENTIFICATION OF SOFTWARE PORTABILITY ISSUES (TASK I)	11
Current Software Environment	11
Target Software Environment	11
Issues of Portability Between the Current and Target Environments	12
DEVELOPMENT OF SOLUTIONS TO PORTABILITY ISSUES (TASK II)	19
Portability Goals	19
Solutions to the Hardware Portability Issues Identified in Task I	20
Solutions to the Software Portability Issues Identified in Task I	26

CONTENTS (Continued)

Page

PROPOSED SCHEDULE OF SOFTWARE CONVERSION ACTIVITIES FOR PORTING THE TWO-HAND TRACKING TEST (TASK III) 31

Activity A: Recompile Existing Code Without Modification 31
Activity B: Develop Primitive Libraries (for Graphics, Pedestal, and Timing Routines) 31
Activity C: Develop Calibration Routines 32
Activity D: Port the Command Interpreter 32
Activity E: Port the Two-Hand Tracking Test 33
Activity F: Design and Implement Software Test Plan 33

APPENDIX A. ORIGINAL COMPUTERIZED TESTING SYSTEM ARCHITECTURE A-1

B. DESCRIPTION OF ORIGINAL SYSTEM PRIMITIVES B-1

C. EXAMPLE OF CONVERSION FROM PASCAL TO C CODE C-1

D. EXAMPLE OF CONVERSION FROM ASSEMBLY TO C CODE D-1

COMPUTERIZED TESTING SYSTEM SOFTWARE CONVERSION AND DEVELOPMENT: IDENTIFYING SOFTWARE AND HARDWARE PORTABILITY ISSUES AND SOLUTIONS

SECTION 1.0
INTRODUCTION

1.1 PURPOSE OF THE RESEARCH

A battery of computerized tests designed to assess psychomotor and other capabilities of new U.S. Army recruits was developed in Project A. Drs. Rosse and Peterson constructed the tests as part of a larger research program to develop and validate selection and classification tests for the U. S. Army Research Institute for the Behavioral and Social Sciences (ARI).

The purpose of this effort is to identify hardware and software portability issues associated with the transition of these tests from the original configuration (early 1980s technology) to currently available hardware/software environments. Hardware issues include items such as changes in the response pedestal, processor, and video. Software issues relate to items such as the compiler, operating system, programming language, and advances in design and documentation.

The need to maintain the psychometric and construct properties of the original tests is an important issue. During porting -- that is, software conversion to other hardware and operating system environments, -- it will be impossible to avoid some changes in test presentation when a substantially different (i.e., more readable) console display will be used. A goal of the porting effort will be to continue measuring the same latent constructs. Changes in item difficulty may be somewhat harder to control, and although there is psychological expertise on the project staff, it will probably be desirable to plan a means to systematically assess the impact of the changes on test/item difficulty.

1.2 ORGANIZATION OF THE PORTABILITY PROJECT

The portability research is being conducted by STATCOM, Inc., as the primary contractor, with the participation of Drs. Rosse and Peterson of the American Institutes for Research, a subcontractor.

The effort is divided into four tasks:

- Task I -- Identification of Hardware and Software Portability Issues
- Task II -- Development of Solutions to Portability Issues
- Task III -- Porting of the Two-Hand Tracking Test

- Task IV -- Porting of the One-Hand Tracking and Target Identification Tests, and Programming Assembling Objects and Figural Reasoning Tests

This report is organized into five sections:

- Section 1 states the purpose of this project and presents some of the background and circumstances of the earlier software activity in Project A.
- Section 2 identifies the portability issues associated with the current hardware and the targeted configurations.
- Section 3 identifies the relevant software portability issues in the proposed conversion from Microsoft Pascal/8088 Assembly language to C language.
- Section 4 contains a discussion of the issues identified and a list of recommendations for conversion of the software to other hardware and operating system environments.
- Section 5 provides a road map of the activities involved in performing the porting effort itself. The Two-Hand Tracking Test will be ported during Task III. The experience acquired in porting this single test will greatly facilitate subsequent porting of other tests in Task IV.

1.3 BACKGROUND

The original Project A programming for the computerized testing system was done in 1985 by Dr. Rosse, a psychologist on the staff developing the series of selection and classification tests. It was written in Microsoft Pascal and 8088 Assembly language, and it was implemented on the then recently introduced IBM personal computer disk operating system (PC/DOS) environment. The experimental test software grew over a period of approximately 2.5 years to meet testing needs that were too difficult to fulfill with the concurrently developed paper-and-pencil tests. When the content was determined to be satisfactory, the computerized tests were combined into a single executable program and administered to experimental subjects in field studies, and subsequent validation activities.

Formulation of functional requirements for the software did not precede the development of the software. Choices of best approaches to software problems were not made at the time, and the approach used in each case was that which was most readily available and which accomplished the immediate needs. Software documentation was minimal because of the experimental nature of the test development effort, which was focused upon ongoing psychological concerns rather than upon software development principles. The present effort will focus on software development and conversion issues.

Those interested in a more comprehensive description of the Project A research program should read "An Overview of the Army Selection and Classification Project (Project A)," by J. P. Campbell, Personnel Psychology, 43, 1990.

SECTION 2.0

IDENTIFICATION OF HARDWARE PORTABILITY ISSUES (TASK I)

This section discusses portability issues related to target hardware configurations for the computerized testing system. Detailed discussion of the issues will be followed by identification of possible solutions and their pros and cons.

2.1 CURRENT HARDWARE CONFIGURATION

The current system was implemented on the following hardware configuration:

- Seequa PC compatible (8088 processor)
- Monochrome high-resolution monitor (640 x 200)
- Two floppy disk drives (1 low density and 1 high density)
- Project A response pedestal

8088 CPU: The Seequa computer used the Intel 8088 chip as its central processor unit (CPU), running at 5 MHz. This was essentially the same speed as the contemporary IBM PC and compatibles, which ran at approximately 4.77 MHz. All of the software is upwardly compatible with the 80286, 80386, and 80486 chips if used with the same peripheral environment.

Console Screen: Project A used an early composite monitor (monochrome green) with an IBM color graphics adapter (CGA). Text mode dimensions were 25 rows by 80 columns of characters. The tests used only the high-resolution graphics mode, consisting of 200 rows and 640 columns of pixels. The cathode ray tube (CRT) screen was rectangular, measuring 6.5 by 7.5 inches.

Response Pedestal and Controller Board: The software in Project A interacted with the response pedestal via a controller board which was inserted into an expansion slot in the computer. The controller board was a modified version of the IBM game board commercially distributed for public use. On the Project A controller board the number of analog inputs and binary (button) inputs was increased from four to eight to accommodate the number of devices built into the Project A response pedestal.

The joysticks, sliding adjusters, and rotary dial presented analog inputs to the software. There were seven such inputs: two for each of the two joysticks (horizontal and vertical directions), one each for the two sliding adjusters, and one for the rotary dial. Analog to digital conversion of the input required software that used the CPU. Specifically, the position of each device was indicated by the resistance measured by a potentiometer attached to the device. The resistance controlled the time interval of a "one-shot" timer circuit on the controller board. The digital value representing the

device position was obtained by a software loop (written in Assembly language) that counted the number of times the loop was repeated until the one-shot timer completed the interval. This method provided differential count values (i.e., device positions) that were reliable for a particular computer and response pedestal combination. Each computer and pedestal combination had to be calibrated before use in testing.

Buttons provided binary input to the response pedestal controller board. There were seven inputs in all: (a) left green, (b) right green, (c) left red, (d) right red, (e) blue, (f) yellow, and (g) white. There were two green buttons on the left and two more on the right; for each pair both buttons had to be depressed in order to register an input.

All buttons were single-pole with normally open switches that provided a binary on/off value to the software. The switch circuits were not debounced. Therefore, a single press of the button would produce several hundred on/off cycles to the software because of the mechanical "scratching" of the electrical contacts. Thus, debouncing had to be accomplished in the software to prevent spurious closures of a key or switch from being recognized as input. This was done by introducing time delays that gave the switch contacts time to settle down.

Real-Time Clock/Timer: The original computerized testing system made extensive use of a real-time clock chip, a National Semiconductor MM58167A. The chip permitted access to its counting registers, which incremented every millisecond, and operated independently of the CPU. Thus, it was possible to time the presentation of stimulus events and response latencies to one millisecond accuracy.

Disk Drives: The Seequa computer used in the testing had two floppy disk drives: (a) A-drive (double density: 360 KB), and (b) C-drive (quad density: 1.2 MB). All programs, batch files, controlling program directives (stimulus files), and data were placed onto two compatible floppy disks.

2.2 TARGET HARDWARE CONFIGURATIONS

This section outlines ARI's general hardware specifications for the revised computerized testing system. The specifications of the target hardware are as follows:

- 386/SX CPU processor running at 16 MHz (minimum)
- Super VGA monochrome video device
- RGI response pedestal (developed by RGI, Inc. [full company name])

2.3 ISSUES OF PORTABILITY BETWEEN THE CURRENT AND TARGET CONFIGURATIONS

The principal portability issues in changing to the target hardware are:

- Change in the response pedestal (Issue H.1)
- Real-time clock for timing needs (Issue H.2)
- Disk-drive change (optional) (Issue H.3)
- Change to higher resolution video adapter (Issue H.4)
- Change to the 386 processor, and increase in clock speed to 16 MHz (Issue H.5)

The next five sections describe each of these issues and their implications for portability.

2.3.1 Change in Response Pedestal (Issue H.1)

Externally, the standard RGI response pedestal differs from the pedestal used in Project A in three ways: (a) it is one inch narrower, (b) it does not have the rotary dial, and (c) it has an additional key marked "HELP."

These external differences do not affect portability directly but are specified here because they may have implications for psychometric and construct equivalence. The lack of a rotary dial in the existing RGI device requires the selection of a substitute means of obtaining some information from test subjects. The addition of the "HELP" button would require a decision on whether and how to use it in the software.

Internally, the RGI response pedestal differs from the original Project A pedestal. First, the RGI pedestal requires an external power source. Second, it has internal processing of analog-to-digital conversion of the analog device. Finally, its communication through a port address differs from the original pedestal.

2.3.2 Real-Time Clock Performance (Issue H.2)

The original test software is precisely timed to millisecond accuracy. The current approach to obtaining the desired timing accuracy will have to be modified in the transition because the on-board real-time clock in the target 386 and 486 computers does not routinely provide the same information as that supplied by the earlier add-on clock. Further study may reveal ways to use the on-board clock. The decision of which clock to use may have an effect on future portability.

2.3.3 Disk-Drive Change (Issue H.3)

The Project A hardware used two floppy disk drives to hold the software, stimulus data, and response data. Three original concerns determined which drives were used and which data they stored.

First, concern about security of the test dictated that software and stimulus data be maintained on floppy disks so they could be removed easily and placed in secure storage. Now that removable high storage capacity and fast access media are available, concern for security no longer needs to relegate the software and stimulus data to floppy disks.

A second concern was making sure that the data collected from the examiner were not lost due to either loss of electrical power or flaws in storage media. This concern required that the data collected from the examinee be stored simultaneously on a second disk.

A third concern was ensuring privacy for the examinees. This required maintaining all examinee data secured on removable disks.

These concerns in concert with the characteristics of drives now available will form the basis for choosing which drives will be used to store the programs and data.

2.3.4 Change in Video Adapter (Issue H.4)

Changing the video adapter will require modifications in the graphics routines, and the proposed higher resolution may affect psychometric equivalence (discussed in Section 3.3.6). The graphics routines will have to be rewritten from the current assembler routines to the third-generation language (i.e., C).

The original console display on the Seequa computer had one odd characteristic. When two or more adjacent pixels on the same row of pixels were set at the same time, the combination created an anomalous "shining" spot. These spots needed to be eliminated in the drawing of figures because they resulted in unplanned (and unacceptable) cues to the respondent during testing. They were eliminated in the original graphics software by not setting two adjacent pixels on the same line. This adjustment would not be needed for the proposed video adapter. Accordingly, the code that made the adjustment would need to be taken out during the port.

Video graphics technology has been changing rapidly since microcomputers first appeared. One can reasonably expect more changes and advances to emerge. Such potential changes and advances should be considered when porting of the tests is being planned.

2.3.5 Change in Processor and Clock Speed (Issue H.5)

Three concerns were identified with regard to changing the processor (CPU) and its speed. First, because of the slow speed of the original hardware, Assembly routines were used for the video, timing, and response pedestal to maximize the execution speed of the software. In the proposed minimum configuration (i.e., a minimum of a 386/SX 16 MHz CPU), the execution speed of routines programmed in C will be more than adequate. As a result, the question arises as to whether any routines should be programmed in Assembly.

Second, although the minimum configuration will provide adequate execution speed, different CPU and CPU speeds would cause the software to behave differently if, for example, video timing was dependent on CPU speed. In the original Project A computerized testing system, this issue was resolved by using the timing mechanism to control the timing of the video presentation. The same method could be used in the new software to address this problem. At most, this method would affect the timing of the video by a small fraction of a millisecond, for the reason outlined below.

Third, using a timing mechanism independent of the CPU still requires the computer to obtain the time information from that device (e.g., from the add-on clock or variable holding the time value). The speed with which the CPU obtains this information will, to a certain extent, affect the value obtained. For example, a slower CPU will take longer to query the timing mechanism, and it is more likely that another millisecond may have ticked. Obviously the faster CPU is more accurate. Thus, the magnitude of difference in querying the timing device across varying CPU speeds needs to be examined.

2.4 ISSUES OF PORTABILITY TO OTHER HARDWARE CONFIGURATIONS

As mentioned above, once the port to the specified hardware configuration has been made, the system will work on virtually all 386 and 486 PCs currently on the market, as long as the implementation of C chosen supports them and the RGI response pedestal and other peripheral devices (e.g., video) do not change.

Once the system has been written in C, it would be portable to other hardware platforms that support the C programming language. In particular, porting to UNIX workstations should not be difficult. Only the pedestal, graphics, and timing routines (i.e., because they may need to access a different clock device) might need to be reprogrammed. Since parameters will be established for these sets of routines, they will be easily identified. Neither set contains many routines. Only a modest effort would be needed to port to a Sun or other UNIX workstation.

In a multiprocessing environment time allocation would be an issue during time-sensitive testing. Appropriate routines would have to be identified in the ported system so that, if the system were ported to a multiprocessing environment, programmers could take explicit steps to control the processor. This would allow time-sensitive routines to run uninterrupted.

SECTION 3.0

IDENTIFICATION OF SOFTWARE PORTABILITY ISSUES (TASK I)

This section discusses portability issues related to the software environment. It describes the target software specifications and discusses, in detail, the considerations and options that will be involved in porting.

3.1 CURRENT SOFTWARE ENVIRONMENT

The software environment for the original computerized testing system was as follows (Appendix A depicts the original Computerized Testing System architecture):

- Microsoft Macro Assembler
- Microsoft Pascal
- Microsoft Disk Operating System (MS-DOS) Version 2.11

Compiler and Assembler: The original testing system was written using Microsoft Pascal (version not available--distributed in early 1984) and Microsoft Macro Assembler (MASM). The original code will compile, assemble, link, and execute correctly using more recent versions of the Microsoft Pascal, MASM, and LINK utility. This is significant because one strategy in porting the original computerized testing system would be to first port the existing software "as is" to the specified hardware environment, before converting the code to C. This would not involve a major effort. It might actually facilitate the recoding because it would provide an existing point of reference on the new hardware that works.

Operating System: The operating system, Microsoft DOS version 2.11, uses batch files (".BAT") extensively to control program execution. The DOS command structure is fully upward compatible with current versions of DOS (3.1 and higher), although newer versions of DOS and language implementations suggest that superior approaches to sequence control could be developed.

3.2 TARGET SOFTWARE ENVIRONMENT

The target software specifications are as follows:

- Borland C as the programming language
- MS-DOS 3.1 and higher for the operating system

There are no portability issues involved in the use of this software environment on the hardware specified in Section 2.2. Borland C can compile a program written for any PC running MS-DOS 3.1 or higher.

3.3 ISSUES OF PORTABILITY BETWEEN THE CURRENT AND TARGET ENVIRONMENTS

The software issues in making this transition are:

- Conversion of Pascal and Assembly code to C (Borland C) (Issue S.1)
- Hardware-dependent routines (Issue S.2)
- Calibration of hardware (Issue S.3)
- Lack of original computerized testing system documentation (Issue S.4)
- Lack of structured design in original computerized testing system (Issue S.5)
- Maintenance of construct and psychometric equivalence

The next six sections describe each of the above issues in detail.

3.3.1 Conversion of Pascal and Assembly Code to Borland C (Issue S.1)

Two types of strategies can be used in porting software from one computer language to another:

- Literal translation
- Conceptual conversion

Literal translation would involve the direct conversion of lines of code written in Pascal to lines of code in C language so that the new program code performs the same computer operations in the same order. For instance, the code for computing a simple equation written in Pascal may be:

```
hypotenuse := sqrt( sqr(a) + sqr(b) );.
```

The identical operations may be accomplished by the following code written for a C language program:

```
hypotenuse = sqrt( a*a + b*b );.
```

In this example, both languages should obtain the same numerical result because both perform the operations in the same functional manner.

However, in some cases translation from one language to another is less direct. For instance, the operations involving character strings in Pascal are very different from those in C because the format for storing them in memory is unique to each of these two

languages. Although both use a byte array of memory locations, Pascal uses the first byte for a character count and stores the ASCII code of the first character in the second byte. C stores the first character in the first byte and terminates the string with a null byte (zero code).

This format difference is important because the syntactical forms used for manipulating strings in Pascal cannot be directly translated into C without interpretation by a programmer. For example, the Pascal programmer may obtain the length of a string by the statement,

```
ncharhead := ord( heading[0] );
```

where the contents of the first (zero-th) byte of the string heading are stored in the integer variable, **ncharhead**. On the other hand, the C programmer has to actually count the number of character codes in the string up to the terminating null code. This can be done in several ways but the method of choice is to invoke the standard C runtime library routine, **strlen**, as follows:

```
ncharhead = strlen( heading );.
```

This particular format difference is important in the present porting effort because the test software to be ported contains a great deal of character string manipulation.

Conceptual conversion would involve duplicating the function of the original software without necessarily duplicating its specific computer operations. Such a conversion rests on a careful study of the function and intention of the software, and requires a thorough understanding of the original computer architecture and the computer languages used. The new software can be produced so that it accomplishes the same functions but does not necessarily follow the same logic used in the original implementation.

A conceptual conversion strategy is essential for porting the Assembly language primitives of the original testing system because there is no way to literally translate Assembly language code into another language. Primitives are small, basic software building blocks. The porting programmer must understand the functions performed by the Assembly language and construct new code using the syntactical forms of the new language to accomplish the same functions.

A principal advantage of the conceptual conversion strategy is that it yields better organized and documented software. This occurs because programmers performing the conversion will have a more thorough understanding of the operations than if they simply translated the code.

Another advantage is that the programmer using a conceptual conversion strategy can use the strengths of the targeted language more freely and effectively. As a result, the original software may be improved upon in the conversion.

3.3.2 Hardware-Dependent Routines (Issue S.2)

In certain parts of the software, code that is specific to the hardware platform must be written. This will be necessary because with the hardware changes, hardware-dependent instructions will be considerably different from those for the original hardware. The pedestal timing and graphics routines will be the most dependent on low-level functions specific to the needs of the hardware. It is important to design the ported software to minimize the impact on future portability to new hardware.

3.3.3 Calibration of Response Pedestal and CPU (Issue S.3)

The original testing system incorporated a pretesting sequence that included testing the timer and the buttons, adjusting the screen to have a 1:1 vertical-horizontal ratio, calibrating analog devices (joysticks, sliding adjusters, and dial), and requesting needed information. The pretesting sequence was performed before each testing session, and a one-line record of calibration constants and other information was written to a disk file. The calibration constants were used to adjust the presentation of test items to the examiner.

The issues are: First, whether such calibration tests will be needed under the new hardware/software environment; and second, if they are necessary, what parameters are involved and how they are affected by the change in the hardware/software environment.

The complete specifications for pretesting in the new environment will necessarily depend upon experience with the new pedestal, new computers, and new testing software. However, it is expected that pretests of each of the following types will be required:

(1) Button function: Test each button to assure that it is opening and closing correctly.

(2) Video:

  (a) Test to assure that the video screen has correct dimensions in both graphical and textual modes, and that in the graphics mode the horizontal to vertical aspect ratio is correct so that a horizontal line n inches long is drawn the same length vertically.

  (b) Test to assure that drawing of video images can be done with sufficient speed to meet software requirements.

(3) Joysticks and sliding adjusters: Test to assure that each of the devices responds smoothly and linearly to movement and that the devices measure movement according to a correct (or correctable) metric.

3.3.4 Lack of Original Testing System Documentation (Issue S.4)

Examination of the original source code indicates that documentation of functions and processes is very limited. Although it is written primarily in Pascal (which is often touted as a self-documenting language), there are not enough comments to readily reveal the organization of the software. Also, the Assembly language primitives are not sufficiently documented to allow an understanding of the details of the methods used in the code.

3.3.5 Lack of Functional Specifications (Issue S.5)

The original testing system had no functional specifications. The original software and hardware configurations were developed over several years of experimentation. Most of the tests were developed and revised as separate executables over a period of 2.5 years. The final executable module of the computerized testing system consisted of a collection of these test modules that were simply combined into a single executable module.

Some effort was made to standardize the software components but some redundancy and some fragmentation of component function still exist. For instance, the parsing of commands was not accomplished by any particular module and tended to be inserted into the code as it was needed.

The issue that surfaces from the lack of clearly defined functional specifications is that programmers do not have a detailed layout of how the system is supposed to behave. Thus, to duplicate a component function, the programmers must examine the performance of the software in its original hardware environment, as well as, carefully examine the Pascal code. This is necessary to determine how the user actually sees the process (i.e., how the operations are sequenced and timed). The executable version has to be examined because interpretation of the code alone can be difficult or impossible where constants and parameters of the software/hardware environment are not clearly defined.

For example, the original software was written with the expectation that it would be used only with the IBM CGA video adapter, and that specific values of the parameters of that particular device could be assumed throughout the Pascal code. Accordingly, the numeric constant, 200, was frequently used in the code because it represented the number of lines of pixels on the CGA display in high-resolution graphics. If the porting programmer is not aware of the purpose of the constant, he or she may use it directly in code intended for a Super VGA display that has 768 rows of pixels.

The use of this numeric constant would not be a problem if this was the only such occurrence, but many such constants were used throughout the code. This makes it necessary for the programmer to track down all constants in the code.

3.3.6 Maintenance of Construct and Psychometric Equivalence

It is difficult, perhaps impossible, to relocate the tests developed for the Seequa computer onto an advanced computer with Super VGA and a modified response pedestal while retaining precisely the same test characteristics. For example, (a) the text will be much more readable and this could affect performance on tests that require reading, (b) the superior graphics resolution will enable figures to be presented more clearly on the screen, (c) the screen will be larger and positioned differently, and (d) the response pedestal has more responsive joysticks and is one inch narrower.

It will be the goal of the conversion effort to make the new version of the tests functionally equivalent to the original tests with regard to construct validity and psychometric properties. Every attempt will be made to keep the tests as closely as possible to the original specifications. Further examination of this issue, however, is beyond the scope of task and will not be addressed in the conversion effort. Additional studies using the converted software will be necessary to establish the construct and psychometric equivalence of the ported tests with respect to the original tests.

There are at least two issues that will need to be addressed by future research: a) item difficulty level, and (b) the latent constructs which the tests purport to measure. Presumably, both issues will eventually be addressed by subsequent research activity to provide normative and correlational information with regard to the constructs. Adjustments might then be made to the converted software to attain psychometric and construct equivalence to the original tests.

There are various approaches to the necessary research. Some are more stringent than others. Below is a sampling of various approaches that could be used, ordered from less to more stringent:

(1) Examine the "look-and-feel" properties of the original tests using individuals with expertise in psychometric issues, familiar with the constructs that the original tests measured, and experienced in using the tests.

(2) Collect distributional data on the converted tests and compare to existing data on the original tests.

(3) Administer both the original and the converted tests (counter-balancing for order of administration) and examine distributional differences. In addition, administer construct marker tests and examine patterns of correlations between the original and converted tests.

SECTION 4.0

DEVELOPMENT OF SOLUTIONS TO PORTABILITY ISSUES (TASK II)

This section sets forth the major goals for successful porting, and then discusses in detail the solutions proposed to meet the hardware and software portability issues identified in Task I.

4.1 PORTABILITY GOALS

Three major goals can be identified for successful porting of the Project A computerized testing system. The elements related to meeting each goal will be listed. These major aims for the proposed porting effort are:

- Hardware and operating system independence.
- Adherence to software engineering principles.
- Construct and psychometric equivalence for the new hardware.

4.1.1 Hardware and Operating System Independence

It will be crucial for future ports of the computerized testing system to have the new system avoid code written for a specific hardware platform or operating system. Certain components of the system, however, do require code specific to the hardware and operating system (i.e., graphics timing and response pedestal routines). Fortunately, there are techniques for designing these routines that can be used to minimize the impact on future ports. The major procedures which will be used to achieve hardware/operating system independence are:

- Code entirely in third-generation programming language (C).
- Minimize compiler-specific routines (use American National Standards Institute (ANSI) routines).
- Use primitives for portions of the code that must be hardware-dependent.

4.1.2 Adherence to Software Engineering Principles

During the porting process, the design and implementation of the system should emphasize adherence to sound software engineering principles. Those that have particular relevance to the kinds of problems encountered in porting are:

- Modularity of design and implementation.

- Use of structured programming techniques.

- System documentation.

4.1.3 Construct and Psychometric Equivalence for the New Hardware

While the look-and-feel of the new system will differ from the original, it is important to minimize the impact of these changes. The following approaches during the porting process will help achieve this goal:

- Maintain the same relative size of the text and graphics.

- Maintain the same timing characteristics for the stimuli.

- Use a single color to imitate original monochrome (i.e., black-and-white) monitor.

4.2 SOLUTIONS TO THE HARDWARE PORTABILITY ISSUES IDENTIFIED IN TASK I

This section identifies alternative solutions for each hardware issue identified in Task I. In some cases only one reasonable solution exists. In all cases the circumstances and reasoning leading to the proposed solution(s) are discussed.

4.2.1 Solution to Issue H.1: Change in Response Pedestal

The original Project A response pedestal communicated with the software through addressable input/output (I/O) ports. The RGI version of the response pedestal is more sophisticated and an alternative means of communication will be necessary. A software driver and a controller board (to be placed into an expansion slot of each computer on which the ported testing system will run) will be required. Buttons that appear in the proposed RGI pedestal but are not needed by the tests, such as the "HELP" button, would simply be inactivated.

Our initial investigation suggests that the Personal 488/AT card and port from IOtech, Inc. with driver software may accomplish the desired interface. The Personal 488/AT converts IBM PC/ATs and compatibles into high-performance Institute of Electrical and Electronics Engineers (IEEE) 488.2 compliant controllers. This package includes enhanced Driver488 software and an AT488 half-slot board for 8- or 16-bit slots, and 1 MB/sec direct memory access (DMA) data transfer. A second option is similar package from National Instruments.

Recommendation: Use a controller board and driver software to interface with the RGI response pedestal. In addition, continue to maintain compatibility with the original Project A response pedestal.

4.2.2 Solution to Issue H.2: Real-Time Clock Performance

The computerized testing system requires a resolution of one millisecond (or better). At least three options for providing millisecond timing to the ported software exist:

(1) Use the same add-on timer chip (MM58167A) that performed the timing function in the original software.

(2) Program the serial communications device to provide countable interrupts at the desired frequency.

(3) Program the 8253 timer chip (on-board virtually all PCs) latch to provide countable interrupts at the desired frequency.

With regard to the first option, the timing chip that performed the timing functions for the original software was described in Section 2.1. It is an inexpensive chip that could be supplementally used in the ported test environment provided that it is disassociated from the interrupt bus, since the timing mechanism on the mother board is controlling the interrupt bus. This option has the disadvantage of requiring hardware additions and changes to the computers that will be used for presenting the ported testing system.

The primary advantage of using this separate clock chip (as in the original software) is that it operates independently from the CPU. Historically this was critical because the Seequa computer's CPU was so slow that using it to assist with timing presented a substantial risk of conflict with functions that depended on the CPU (i.e., loops for analog/digital (A/D) conversion for establishing the positions of the sliding adjusters on the response pedestal). Conflict would have arisen if the time-critical functions were operating when the CPU was needed to perform the timing function. Because time-critical functions would no longer be a problem in the target hardware environment, the use of a separate clock chip does not appear to be as essential as it was in the past.

The second option, based on programming a serial communications device to provide countable interrupts, is also viable. A standard serial communications device has a clock running at 1.8432 Mhz to control the rate at which it sends and receives serial data (i.e., the baud rate in bits per second). This rate can be programmed by providing a divisor of the base frequency. Also, the device is capable of generating hardware

interrupts (COM1 or COM2) for conditions of "data available" or "holding register empty."

A timer for the test software could provide an interrupt service routine which (a) counts the "holding register empty" interrupts, and (b) places another byte into the holding register after each interrupt so as to cyclically generate the next interrupt. Thus, the interrupt count on successive occasions would provide an accurate measure of the elapsed time. Using this approach with a timing precision of one millisecond requires the CPU to service at least 1000 interrupts per second. This capability will not be a problem when the minimum equipment used is a 16 MHz 386/SX processor.

The third option is similar to the second. It consists of programming the 8253 timer chip latch to provide countable interrupts at the desired frequency. This option is also viable. The 8253 (or 8254) timer chip, a standard component of computers in the target class, could be used like the serial communications device just described. The 8253 has a base frequency of 1.19318 MHz which the software can divide by setting a count-down latch. This device also generates hardware interrupts at the programmed frequency. Using this device for timing has essentially the same characteristics as the method described for the serial communications device; it would require that a minimum of 1000 interrupts per second be serviced and counted by an interrupt service routine in the software.

It is important to note that changing the 8253 timer latch introduces some additional, but not prohibitive, software considerations as compared to using the serial communications device option. The interrupts generated by the 8253 timer chip are used by the computer for other purposes. Its normal interrupt rate (i.e., 55 milliseconds between interrupts) is used in maintaining the calendar clock and in I/O disk functions (in the original IBM PCs the timer was also used for video timing). These functions are implemented by an interrupt service routine provided with the MS-DOS system. Thus, if the cyclic interrupt frequency generated by the 8253 timer is increased and one wants to maintain compatibility with other system resources, a replacement interrupt service routine for the test software would be necessary. While the original service routine updates the system clock with each timer click, the replacement routine would only do this approximately once every 55 times that the timer ticked (i.e., maintain default timing characteristics).

To examine the feasibility of programming the 8253 timer latch, some experiments were performed with prototype software that could be used in the ported test software. The purpose was to determine whether timing of sufficient accuracy could be achieved while maintaining sufficient processing capability for performing other functions. The prototype timing routine allowed initialization of the 8253 timer with a latch value within the permissible range of 0 to 65535. (Note: A value of 0 is the same as 65536, i.e., the default latch value.) The experimental interrupt service routine serviced the interrupts from the 8253 timer chip by (a) counting the interrupts, and (b) executing the system's

original interrupt service routine at intervals of approximately 55 milliseconds. In tests of the experimental service routine using a 16 MHz 386/SX computer, a latch value that generated as many as 9,861 (only 1,000 are needed for millisecond timing) interrupts per second did not interfere with ongoing I/O functions, such as keyboard input and the reading and writing of a 32 KB text file.

Thus, both the second and the third options maintain a cyclic interrupt count to provide timing to the software. Two questions affecting accuracy need to be considered. First, in the case of the second and third options, how quickly is the timing count updated? Data presented above indicate that for the third option the count can be updated more than 9,861 times per second without "lock-up." This means that it takes, at most, 101/1000ths of one millisecond to update the count. Second, tests showed that the count could be accessed more than 250 times per millisecond (i.e., 4/1000ths of one millisecond). Thus, it takes much longer to update the count than it does to read it, and the average total error in the time read is, at most, 105/1000ths of one millisecond. With faster machines, this error will be even less.

Recommendation: Adopt the third option, program the on-board 8253 timer chip. This option is the most appropriate because it (a) is accurate to within one millisecond, (b) does not interfere with other computer operations, and (c) does not require the computer to have extra devices, such as a supplementary timer chip or a serial communications device. Its use is recommended to fulfill the timing requirements of the ported software.

4.2.3 Solution to Issue H.3: Disk-Drive Change

The software to be ported will be designed to accommodate the use of hard disk, RAM disk, and floppy disk drives. The original configuration used only two floppy disk drives.

Numerous options exist for assigning the various storage needs to specific disk drives, but data and program security and integrity concerns outlined in Section 2.3.3 currently establish some limitations. Since no data or programs may be maintained in permanent memory in the computer (i.e., because it is not possible at this time to be certain that the computers will be stored in a secure location), a hard disk drive cannot be used to store this material. Thus, at present all programs and data will have to be read from floppy disks and examinee data will have to be stored on floppy disks.

Note that this requirement does not preclude the software from using RAM or the hard disk drive to store the programs or program data, provided that the material is completely erased from the computer at the conclusion of the testing session. The examinee data integrity requirement, however, precludes storage of examinee data on a RAM drive.

Recommendation: Allow test administrator to select which drives will store examinee data, backup data, and from where software will retrieve necessary test information.

4.2.4 Solutions to Issue H.4: Change in Video Adapter

The concerns related to the video adapter are those of segmenting the video routines, using primitives (small, basic library routines which perform operations that are hardware specific), and assessing the visual impact of these primitives.

Well-designed primitives must address (a) the purposes of the software, and (b) the special requirements of the hardware system for a particular implementation. With such well-designed primitives, minimum changes will be needed when the tests are implemented on different hardware. It is therefore necessary to define each primitive clearly in terms of how it provides the linkage between the application and a suitable domain of potential hardware systems. Ideally, the domain should include not only presently known computer systems, but also the anticipated future properties of video graphics adapters.

Recommendation: For future upgradability, use video primitives to compartmentalize code dependent on the video adapter.

4.2.5 Solution to Issue H.5: Change in Processor and Clock Speed

Three concerns were identified in Section 2.3.5 with respect to the change in processor and clock speed.

First, the increase in speed in the target equipment eliminates the need for Assembly routines. Only C language routines should be used in the conversion.

Second, critical events such as video presentation should not depend on the speed of the processor. Fortunately, the original software used code designed to be independent of the processor speed. It appears desirable to continue using the approach used in the original software. An example of the typical method for controlling timed events is described below. For this example, abstracted from the One-Hand Tracking Test, the programmer's objective is to control the rate at which a target moves and the rate at which a cross-hair controlled by the examiner is moved.

In this example, the exact number of time intervals that need to elapse during the presentation of any particular test item is known. The number of intervals is determined from the length of the target path. The length of each interval is controlled (within minimum limits) by using a parameter in a stimulus file.

Within each interval there is a set of operations that must be accomplished. The amount of computer time required to complete these operations varies from one interval

to the next. The method used in the original tests makes use of the millisecond timer. This same approach could be used in porting the code.

An iterative loop is used to execute the required operations. Before the loop starts, the millisecond timer is reset (to zero). The following steps are completed for each interval:

(1) The target position is computed.

(2) The trace line trailing behind the target is erased.

(3) The joystick position is read from the response pedestal.

(4) The new cross-hair position is computed from the joystick position.

(5) The cross-hair at its previous position is erased from the screen and a cross-hair is drawn at the new position (unless the cross-hair position is "frozen" when the computed position exceeds the bounds of the display).

(6) The target (box) at the previous position is erased and a target is drawn at the new position.

(7) The squared distance of the cross-hair to the center of the box is computed and accumulated (toward the computation of the root-mean-square score for the item at completion).

(8) The millisecond timer is read repeatedly (and all operations are paused) until the specified time (from last interval) has elapsed.

(9) Finally, an error is noted if the elapsed time exceeds the specified interval.

Thus, operation (8) assures that the intervals will end in the specified number of milliseconds, provided that operations (1) through (7) do not take longer than the interval.

The above method of controlling timed events during the presentation is used in the existing software. In the original testing system all activities requiring precise timing (i.e., presentation of stimuli) were programmed with millisecond accuracy. Successful porting from Pascal to C requires only that the approach be duplicated and that it can execute on the target computer within the time interval specified.

The original software invoked an error detection system that detected and noted in the data file any failure of the software to execute within the specified time. The same error detection capabilities will be incorporated in the ported software.

A third concern identified in Section 2.3.5 also involved the speed of the CPU. Namely, that different speed CPUs could indirectly affect timing by requiring a different amount of time to update and access the time. However, if the timing method recommended in Section 4.2.2 is used, this problem is resolved, since a minimally configured machine (i.e., 16 MHz 386/SX) can update and access the time with an error of, at most, 105/1000ths of one millisecond. This means that a slightly faster processor would yield a value differing by 1 millisecond, at most, 105 times out of every 1000 times. Thus, the largest timing error that could be expected across different CPU speeds is 105/1000ths of one millisecond. This is a small amount of error due to CPU speed.

Recommendation: Use the approach employed in the original testing system in the new software to enable the video presentation to be independent of the speed of the processor. Additionally, use only C language routines throughout the conversion.

4.3 SOLUTIONS TO THE SOFTWARE PORTABILITY ISSUES IDENTIFIED IN TASK I

This section identifies the alternative solutions for each software issue identified in Section 3.3. In some cases only one obvious solution exists. Each solution is described and discussed.

4.3.1 Solution to Issue S.1: Conversion of Pascal/Assembly to Borland C

A major issue identified in Section 3.3 related to converting Pascal and Assembly routines to C, and involved choosing the conversion method: a literal conversion versus a conceptual conversion. The advantages and disadvantages of using each method are discussed below.

Literal Conversion. As mentioned in Section 3.3.1, literal conversion involves a line-by-line translation of the Pascal and Assembly code to C code. There are two ways to accomplish this task: manual and automated. In the manual method programmers read the lines of Pascal and Assembly code and replace them with the equivalent C statements. Automated tools are available to assist in the conversion process.

A Pascal to C translator by Milton Brown Software, Inc. was reviewed in depth for this project. This automated package appears to handle most of the constructs used in the original code. Use of this tool would increase the rate at which the conversion could be accomplished.

The advantage of the manual method over the automated method is that it reduces the chance of errors during the conversion. Its major disadvantage is that it consumes more time than the automated method. However, since the evolution of automated translation tools is still at an early stage, considerable review of the code

produced by those tools would still be necessary, which would lessen the time advantage of those automated methods.

Conceptual Conversion. Conceptual conversion involves examining both the source code files and the executable version, determining the required behavior of the system (i.e., identify the functional specifications which are not stated in the Project A documentation), and re-implementing the required functional behavior in the ported system. This method of conversion requires more expertise in the programming languages involved (Pascal, Assembly, and C) than a literal conversion. Also, it requires a very solid understanding of system design techniques. It is the most time consuming of all the methods; even a manual literal conversion is faster.

While an automated literal conversion would be the best alternative from a resource standpoint, the complexity of the software and the lack of original documentation suggest that a conceptual conversion may be wiser. Even though the conceptual approach would take more time, a major advantage is that the new code would be better integrated. The original code was pieced together as requirements unfolded in a bottom-up fashion. However, now that the test system is complete, the port to C can also be used to develop an integrated design. The port could use structured coding techniques, which would make the code more maintainable as well. The following section discusses modularity, which is also a goal of the porting process and which would produce an improved software design.

Finally, as noted in Section 3.3.1, a conceptual conversion strategy is essential for porting the Assembly language primitives because there is no way to literally translate Assembly language code to another language. Thus, the porting programmer must understand the functions performed by Assembly language and develop new code to attain the same functionality.

Recommendation: Use conceptual conversion to port the software. Conceptual conversion has a strong advantage over literal conversion in that the ported software can be better organized and documented than was true for the original software. This is possible because the programmers will have a more thorough understanding of the operations than if they simply translated the code. Another advantage is that the programmer using a conceptual conversion strategy can utilize the strengths of the targeted language more effectively. Thus, the converted software generated through conceptual conversion will represent an improved product.

4.3.2 Solution to Issue S.2: Hardware-Dependent Routines

Some routines that exist in the original program are specific to the hardware on which they were implemented. Such dependencies cannot be avoided. However, a goal of this effort is to produce code that lends itself to further adaptations on other hardware platforms, such as UNIX workstations.

The concern with future portability suggests that library routines which segment the hardware-dependent routines would be beneficial. These basic library routines, called "primitives," are small segmented routines which serve as building blocks for the rest of the code (e.g., **DrawLine**, which simply draws a line between two coordinate points). This modular design separates the hardware-dependent portions of the code so that replacing them in the future does not affect the remaining code.

Fortunately, a library of these routines, mostly written in Assembly language, was developed for the original timing, response pedestal, and video graphics functions. These exist as separate modules and duplicating them in C should be fairly straightforward. We have identified all of the primitives necessary for the ported code in Appendix B.

Recommendation: Isolate hardware-dependent routines and develop libraries for them.

4.3.3 Solution to Issue S.3: Calibration Routines

The original software contained a separate process (named PRETEST.EXE) which provided testing of critical hardware functions and calibration of both video properties and response pedestal readings. The intent was to standardize the visual presentation and readings across response pedestals.

Visual tests were made of the screen. Tests of the buttons, joysticks, and sliding adjusters assured that the equipment was working. Tests of the timer hardware assured timing accuracy. To minimize differences between test presentations across computers and response pedestals, specific response pedestal calibration operations were performed and used by the software.

One important calibration parameter is the ratio of horizontal to vertical units in graphics mode. For example, circles should appear round; horizontal and vertical cross-hair movements should be comparable. Another set of important parameters relates to the calibration of measurement units obtained from the analog devices so that physical movements of joysticks and sliding adjusters on different pedestals can be converted to the same metric by the software.

Recommendation: Incorporate testing and calibration functions in the ported software. Calibration and testing are planned for the specific target environments. For future environments, we will specify any anticipated attributes which should be tested.

4.3.4 Solution to Issue S.4: Lack of Original Computerized Testing System Documentation

Two major issues identified in Task I arise from the lack of original software documentation. The first involves the difficulty in translating the original code to C. The second relates to the documentation requirements imposed on the ported system.

Namely, since there is no starting point, all documentation will have to be developed in entirety.

Documentation requirements include the following:

- In-line code documentation
- Data flow diagrams
- Maintenance and test procedures
- Software programmer's manual
- Software user's manual

The code will be thoroughly documented with function headings that include at least the following: description of the function's behavior, description of all local and global variables used, and identification of other routines which call and are called by the function. Also, there will be additional in-line descriptions of important control sequences and data structures throughout the code.

Data flow diagrams will be produced as part of the porting effort to describe the structure of the program and how the system communicates with its users. A Software Programmer's Manual will be provided describing the ported software, for use by future programmers when conducting maintenance or implementing other tests. This manual will contain a section on maintenance describing how item and instruction scripts for the tests can be created and modified. A Software User's Manual will be provided describing how the program operates and the details of administering the tests to examiners.

Recommendation: Ensure that the conversion effort produces the following documentation: in-line code documentation, and software user's and programmer's manuals.

4.3.5 Solution to Issue S.5: Lack of Functional Specifications for Original Computerized Testing System

Lack of clearly defined functional specifications leaves the programmers without a clear and thorough understanding of a system's functions and behavior. To understand the original testing system, programmers need to examine not only the code but also the performance of the software in its original hardware environment.

Recommendation: Ensure that programmers review the original code thoroughly, and that they have access to an executable version, in order to assess how the user actually "sees" the process, and how the operations are sequenced and timed. In this manner, functional specifications for the new system can be established.

SECTION 5.0

PROPOSED SCHEDULE OF SOFTWARE CONVERSION ACTIVITIES FOR PORTING THE TWO-HAND TRACKING TEST (TASK III)

This section outlines the activities to be accomplished in Task III, the actual porting of one computerized test, the Two-Hand Tracking Test. It is organized into six activities, A through F, to serve as a roadmap for the port process. Appendixes C and D give examples of code conversion required by the porting activity.

5.1 ACTIVITY A: RECOMPILE EXISTING CODE WITHOUT MODIFICATION

The existing code can be recompiled to run on the specified hardware configuration with nominal modification because it was written to run on an IBM PC compatible. This step can be expected to facilitate the port by showing how the original system performs and appears on the new hardware configuration.

5.2 ACTIVITY B: DEVELOP PRIMITIVE LIBRARIES (FOR GRAPHICS, PEDESTAL, AND TIMING ROUTINES)

The primitives can be developed independent of the tests and can operate as libraries. These libraries will include specific routines for graphics, the pedestal, and timing. They can be easily located and modified if the hardware changes in the future.

5.2.1 Activity B.1: Develop Routine Libraries

All software functions, including primitives, will be organized into libraries for use by the various software subroutines. Many of these library routines will be callable from anywhere in the software. Appendix B describes all the primitives that were written in Assembly language for the original computerized testing system.

5.2.2 Activity B.2: Investigate Other Compilers

It is important that the library specification use components that can be replicated on other platforms and other C compilers to assure portability to other operating system platforms and compilers. This review of other compilers will prevent use of features that are found only in Borland C. For example, Microsoft Windows applications are very difficult to port to other platforms because of their unique characteristics. However, Borland's graphics libraries do not share this problem to the same degree. This issue, specifically with respect to a UNIX platform, will be investigated.

5.2.3 Activity B.3: Code Library Routines

All library routines will be coded in Borland C.

5.2.4 Activity B.4: Document Library Routines

All library routines will be documented.

5.2.5 Activity B.5: Test Library Routines

Once the routines are coded, small library test programs will be implemented independent of the main programs to validate their effectiveness. This independent evaluation will increase code accuracy and reduce the complexity of the debugging process.

5.3 ACTIVITY C: DEVELOP CALIBRATION ROUTINES

This activity will port the code that tests all of the parameters necessary for ensuring a homogeneous testing environment for all examiners, regardless of the specific machine used for testing.

5.3.1 Activity C.1: Identify All Calibration Parameters

The preliminary investigation identified the calibration routines that use parameters such as aspect ratio and joystick range. The remaining parameters will be identified.

5.3.2 Activity C.2: Code Calibration Routines

All calibration routines will be coded in Borland C.

5.3.3 Activity C.3: Document Calibration Routines

All calibration routines will be documented.

5.3.4 Activity C.4: Test Calibration Routines

Once the calibration routines are coded, they will be tested on the specified target hardware.

5.4 ACTIVITY D: PORT THE COMMAND INTERPRETER

The command interpreter will be segmented into a separate module and ported to C. This interpreter will be responsible for dispatching all script commands for the tests. The conversion will be tested during the port of the Two-Hand Tracking Test.

5.5 ACTIVITY E: PORT THE TWO-HAND TRACKING TEST

With the building blocks for the tests converted and tested, the primary task remaining is that of porting the code for the Two-Hand Tracking test from Pascal to Borland C.

5.5.1 Activity E.1: Identify and Segment Hardware-dependent Pascal Code for the Two-Hand Tracking Test

Per Section 4.3.2, hardware-dependent code will be identified and segmented.

5.5.2 Activity E.2: Perform Conceptual Conversion of Code to C

Per Section 3.3.1, conceptual conversion of code will be performed.

5.5.3 Activity E.3: Document Ported Code and Produce Appropriate Manuals

Per Section 4.3.4, documentation will consist of in-line code documentation, data flow diagrams, maintenance and test procedures, a software programmer's manual, and a software user's manual.

5.6 ACTIVITY F: DESIGN AND IMPLEMENT SOFTWARE TEST PLAN

A software test plan will be designed and implemented to examine the functionality of the ported software on the target hardware and to verify that the software performs identically across various configurations using Intel 80386 and 80486 CPUs.

APPENDIX A

ORIGINAL COMPUTERIZED TESTING SYSTEM ARCHITECTURE

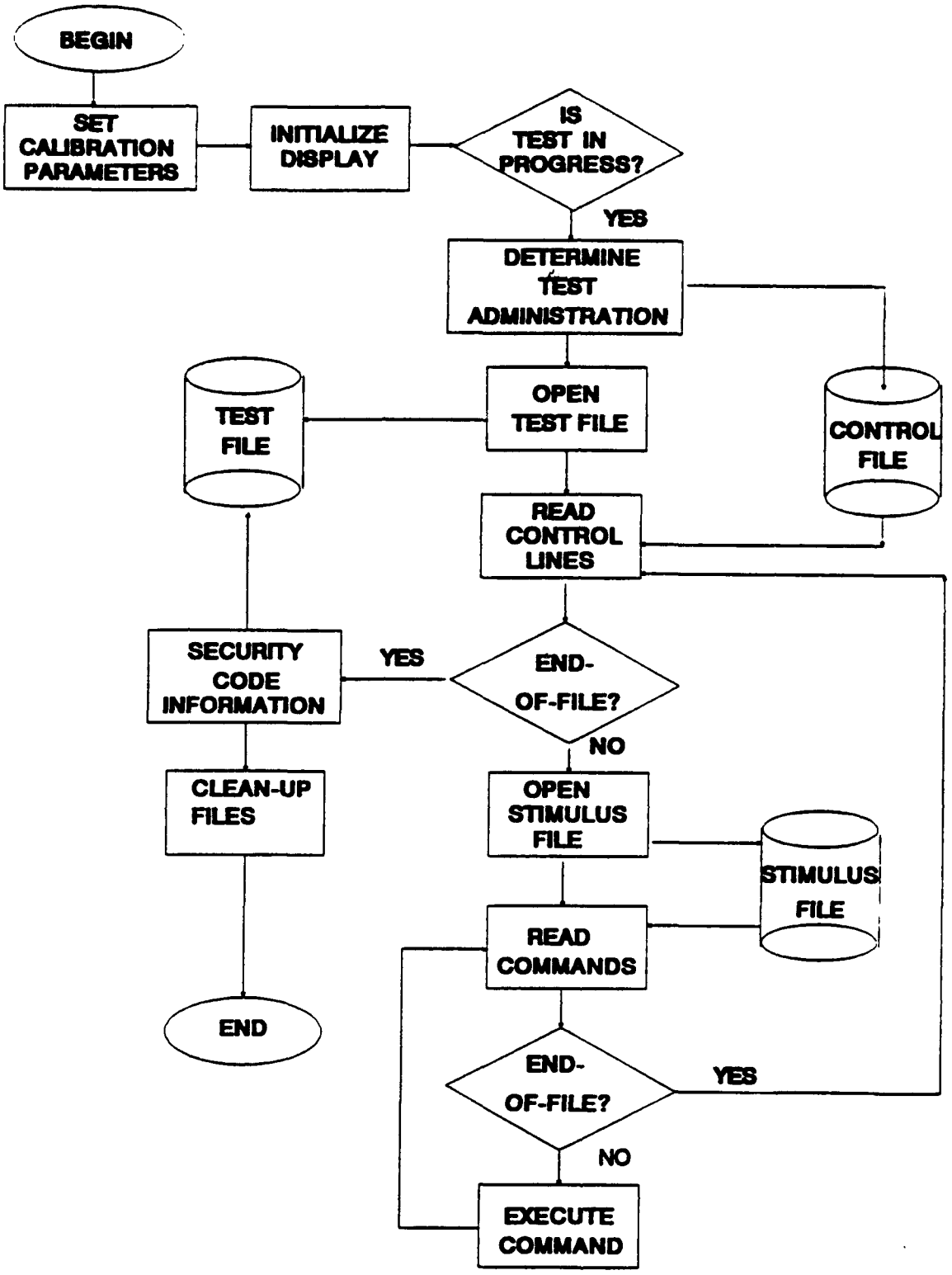

Figure A-1. Data Flow Diagram of Original Computerized Testing System Architecture.

PRETEST PROGRAM

The following describes the detailed steps of the command interpreter for the original computerized testing system:

BEGIN PROGRAM (main).

Set calibration parameters read from file, **CALIBRAT.DTA.**

Initialize display (set to text mode, blank screen).
Look for test in progress (indicated by existence of file **CURRENT.SSN**).

If test in progress then:
display message to call for test administrator to make decisions:

Monitor choices:

(1) Bypass completed tests and resume testing (same respondent).

(2) Process existing data and start new test (different respondent).

If choice (1) then identify respondent and resume testing (loop below begins after last completed test),

else clean up temporary data files to the single, concatenated form and start new respondent at beginning of tests.

Open test-list file **CONTROL** (list of tests to be run).

TESTING LOOP:

Read control lines (test number and stimulus file name) to find next test not completed as indicated by existence of temporary data file for the test.

Open corresponding stimulus file of next test, read and execute all commands (using function command) in the selected stimulus file to end-of-file.

Issue diagnostics if illegal command in stimulus file found.

FINISH UP (test administrator intervention):

Requires access code by test administrator (Button sequence).

Test administrator checks SSN.

Clean up temporary files of test data and aggregate into one file for the respondent for later "downloading" to central data analysis site.

STIMULUS FILE PROCESSING

Function Command (called only from main program)

General: Reads, identifies, and executes the next command line listed in an open stimulus file. Function has value of **"true"** if command is successfully processed, else it has a value of **"false."** Commands are broken into two types: (1) General commands that may appear in any stimulus file, and (2) specific commands for test items that generally appear only in the stimulus file corresponding to a particular test.

General commands which may appear in any stimulus file:

	Command:	Operation:
1.	Blank line	None--permits blank lines in text.
2.	>end	None--used to interrupt command sequences that require more than one line.
3.	>identify	Obtains respondent identification (SSN) and saves it in temporary file, **CURRENT.SSN.** (See function **ident_out** in **DRIVER.PAS.**)
4.	>text ...	Inputs literal text from succeeding lines in stimulus file and displays it on console screen. (See function **text_setup** in **PARS.PAS.**)
5.	>clear ...	Clears (blanks) console screen and places it into normal text mode. (See function **clear** in **PARS.PAS.**)
6.	>on	
7.	>off	Direct control of screen: turn screen on or off. Forces normal text mode if currently in graphics mode. (See function **enable** embedded in function command in **DRIVER.PAS**)
8.	>delay=n	Delays program execution for n time units (each unit is 1/10th of a second). (See function **hold** embedded in function command in **DRIVER.PAS**)

9.	>button=n	
10.	>button=color	Requires respondent to press specific button designated either by number (1-7) or color. (See function **read_button** in **PARS.PAS**)
11.	>display scores	Displays summary of test results of current test on console screen. (This was used only for test demonstration purposes.) (See function display embedded in function command in **DRIVER.PAS**)
12.	>reset data	Resets memory arrays used to accumulate response data to starting status (usually for a new test). (See function **reset_scores** in **OUTMOD.PAS**)
13.	>joystick	Invokes protocol for respondent to select left or right joystick. (See function **setstick** in **OUTMOD.PAS**)
14.	>start timer n	
15.	>stop timer n	Starts and stops timer **n** (n=1,2,3,4). The four timer values are output to the data file using the **">write data"** command below. (See function **setinterv** embedded in function command in **DRIVER.PAS**)
16.	>dial=n	Invokes protocol for respondent to select from list of textual items (previous setup of options by **">text"** commands is required). Respondent uses dial on the response pedestal to select. The parameter **n** is a 3-character item identification for writing the data. Option for flashing (blinking) text provided for displays that do not have text highlighting capability. (See function **dial** in **PARS.PAS**)
17.	>age	Invokes protocol for respondent to enter age using the dial on the response pedestal. (See function **get_age** in **PARS.PAS**)
18.	>normal	Causes console screen to change from graphic mode to text mode (no action if already in text mode).
19.	>cursor	Activates cursor display on console screen.
20.	>cursor hide	Deactivates cursor display on console screen.

21. >cursor=r,c — Activates cursor display and places it at text coordinates indicated by parameters **r** (row) and **c** (column). (See function **cursset** embedded in function command in **DRIVER.PAS**)

22. >write data — Invokes protocols for writing accumulated data to temporary file (ASCII) on disk. Data format and contents of file are determined by which test is in progress.
(See function **writout** in **OUTMOD.PAS**)

23. >rem — No action. Remaining information on the line is ignored. This command is used for comments in stimulus file text.

Commands specific to individual tests:

24. >ident=n — Presents one item of the Identification Test where **n** is a 3-character identification code. Options permit specification of which pre-drawn figures to use, correct response, time limit, pre-trial interval, size of target, rotation of target (in degrees), and direction which the target is to face.
(See function **targ_ident** in **IDENTMOD.PAS**)

25. >gun=n — Presents one item of Cannon Shoot Test where **n** is a 3-character item identification code. Options permit directing the gun icon to specified point coordinates, target placement relative to impact point, specifying distance from gun icon, and distance of travel to optimal "fire" point, angle of target approach, target size, speed (in milliseconds of update), and pre-trial interval.
(See function **pres-cannon** in **CANNMOD.PAS**)

26. >trace=k — This command does not stand alone. It begins presentation of the one or two-hand tracking tests: if k=1 then succeeding "**>path**" commands for items are for the one-hand tracking test; else if k=2 then succeeding "**>path**" commands for items are for the two-hand tracking test. This command is in effect until a command that is not a "**>path**" command is encountered or until "**eof**" (i.e., end-of-file) is encountered in the file.
(See function **trace** in **TRACEMOD.PAS**)

27. >path=n	May only follow a "**>trace**" command. Presents one item for tracking test (indicated on the "**>trace**" command). The parameter, **n**, provides the item identification code for the data file. (See function **setpath** in **TRACEMOD.PAS**)

APPENDIX B

DESCRIPTION OF ORIGINAL SYSTEM PRIMITIVES

A number of primitives written in IBM PC Assembly language evolved throughout the development of the original software for the Project A tests. It is useful to examine and explain each of these primitives in order to plan for porting the software to both a new programming language and new hardware.

The following is an exhaustive list of primitives that were written in Assembly language for the original computerized testing system. To avoid confusion in the discussion, declarative statements for the primitives are written in C language rather than the original declarations in Pascal language. Each primitive is briefly described and discussed.

The primitives are organized into the following categories for purposes of exposition:

A. Interrupt Control Functions
B. Port Access Functions
C. DOS Control Function
D. Response Pedestal Functions
E. Keyboard Function
F. Cursor Control Functions
G. General Video Adapter Functions
H. A Miscellaneous Graphics Function
I. Video Text Functions
J. Video Graphics Functions
K. Timing Functions
L. Timed Delay Functions

A. Interrupt Control Functions:

The following two primitives were used to invoke the hardware commands **cli** and **sti** for clearing or enabling bus interrupts (e.g., video, real-time clock, keyboard):

a. void cli(void);

This function disables bus interrupts by invoking the **"cli"** hardware command.

b. void sti(void);

This function enables interrupts by invoking the **"sti"** hardware command.

Neither of these was called in the final version of the original computerized testing system and they are included here only because they appear in the listings. In the final version of the software (approx. June, 1986) the needed hardware commands were embedded in the Assembly language primitives. Primarily, they disabled the interrupts while servicing input from the analog devices of the response pedestal.

B. Port Access Functions:

The early version of Pascal used in developing the test software did not contain library functions for accessing port addresses. The two functions which accomplish this were written in Assembly language as follows:

a. void outport(char port, char val);

This function writes a byte, **val**, to the port address **port**.

b. unsigned char import(char port);

This function returns the value read from port address **port**.

These two functions can readily be replaced by functions **outportb** and **inportb** in the Borland C runtime library. They will likely be used extensively for communication with the RGI pedestal and the external timer. These are not portable functions since they differ in name and calling sequence across compiler products and software systems. Their usage will be well documented.

C. DOS Control Function:

At this time, it is unclear whether or not this function will need to be ported.

void bootstrap(void);

This function invokes DOS interrupt 0x19 which forces a reboot of the system.

D. Response Pedestal Functions:

The following contains a description of the primitives that are used by the Pascal version of the software for interfacing with the original Project A response pedestal:

a. int stick(int nr);

This function returns the value of the current position of the **nr**-th analog input (nr=1,2,...,7), i.e., nr=1 for the left-joystick horizontal, nr=2 for the vertical, nr=3 for right-joystick horizontal, and so on.

b. int switch(int number);

This function returns the value of **"true"** if the button **number** is pressed, otherwise it returns **"false."** This primitive, though in the source file, is not called in the final version of the original computerized testing system.

c. void buttons(int *value);

This function sets **value** to an integer indicating the current button(s) pressed. The value returned is as follows:

0 = no keys depressed,
1 = home position (all four green buttons),
2 = yellow button,
3 = blue button,
4 = white button,
5 = red (left),
6 = red (right),
7 = left-side home buttons,
8 = right-side home buttons.

Debouncing of buttons was accomplished by requiring identical reads for approximately 1/400th of a second.

Additionally, this primitive disables the video if two or more of the above values are indicated simultaneously, in order to discourage test subjects from pressing buttons in illegal combinations.

d. int buttime(int start, int maxtime, int *time);

This function is used to measure response latencies by measuring the interval of time from (1) the time of call to (2) the point where a button numbered other than **start** is pressed. The parameter **maxtime** is a time limit on how long it will wait for a button press. The elapsed time is returned in the variable **time**. Times are in units of 1/100th of a second.

These functions may remain essentially the same in the ported software. They will have to be redesigned and rewritten for the RGI response pedestal which has an entirely different interface with the test software.

E. Keyboard Function:

The following primitive was required only because the early version of Pascal had no function for detecting a keystroke:

char inkey(int flag);

This function returns a value of 0 if no keystroke is detected; else it returns the character detected. The parameter **flag** specifies whether to return with or without waiting for a keystroke. If **flag** is set to 0 it means not to wait, and if it is set to 1 it means to wait.

If needed, this function is readily replaceable using the runtime library routines **kbhit** and **getch**. For example, the direct substitution could be made as follows:

```
#include <conio.h>

char inkey(int flag)
 {
  if(!kbhit())
   if(!flag)return(0);
    else while(!kbhit());
  return(getch());
 }
```

F. Cursor Control Functions:

The following two primitives also originate from the lack of a modern compiler runtime library:

a. void cursmov(int row, int col);

This function moves the text cursor to specified **row** and **column**.

b. void curshid(int flag);

This function enables/disables cursor (i.e., flag=0 for cursor disabled; else cursor enabled).

Borland C has a direct substitute for the function **cursmov** in the runtime library function **gotoxy**. The second function, **curshid**, is probably not portable since, as currently written, it works only with specific display hardware, i.e, IBM CGA graphics adapter. If needed, it may be possible to use generalized runtime library routines, such

as the function **_setcursortype** which is currently available in the library of the Borland C compiler.

G. General Video Adapter Functions:

These functions control the operational mode of the CGA video adapter. They are as follows:

a. void scenabl(int flag);

This function enables/disables the video display. If the flag is **"true"**, the video display is enabled. If the flag is **"false"**, the video display is disabled. This function is used to hide screen operations from the test subject and control the point in time that the examinee is exposed to it.

b. int enableflg(void);

This function returns the value of **"true"** if screen is currently enabled (using **scenabl**), else it returns **"false."**

c. void gnormal(void);

This function sets the CGA video adapter to normal text mode (80 x 25).

d. void ghigh(void);

This function sets CGA video adapter to high resolution graphics mode (640 x 200 pixels and 80 x 25 character text).

e. void gclear(int color);

This function simply sets all screen pixels to the value of the argument **color**: 0 for black, 1 for white (no action if in text mode).

These functions can be readily rewritten using runtime library routines from the Borland C compiler.

Because runtime library calls for graphics functions are not generally portable across compiler products or computers, the planning and documentation will involve more than simple code substitution.

H. A Miscellaneous Graphics Function:

The original test software contains a peculiar primitive written in Assembly language which requires some explanation. The primitive is as follows:

void blnktarg();

This function blanks the first 72 lines of the graphics screen.

This primitive exists because the target identification test protocol required the blanking of only part of the screen when all four green buttons were released (i.e., the figure displayed at the top of the screen had to disappear). Moreover, the blanking had to be very fast because the software had to complete the blanking before the subject reached another button. Therefore, this optimized primitive was introduced to meet this specific need. Presumably, speed will not be an issue for the ported software on a 386 or 486 computer.

I. Video Text Functions:

All text displayed on the screen (with the CGA adapter in text mode) is placed there by the functions in this group. Although they are readily duplicated by standard C runtime library routines of many compiler products, they were originally written for speed optimization purposes. They are as follows:

a. void scattr(int attribute, int row, int col, int nrow, int ncol);

This function sets character attributes for window (i.e., box) on screen (i.e., the foreground and background colors, intense mode, and blinking). The attribute is set in a window starting at **row, column** coordinates and of size **nrow** number of rows and **ncol** number of columns. The coding of the attribute is a 4-digit (decimal) value where each digit controls an attribute of the characters to be displayed. Each digit has the following meaning:

digit 1 (1000s) = blink flag; 1 to blink, else 0
digit 2 (100s) = 1 for foreground intense, else 0
digit 3 (10s) = background color code
digit 4 (1s) = foreground color code

b. void scstr(int row, int col, int nchar, char *string);

This function writes a character string to the console screen starting at the parameters **row** and **col.** The character string contains **nchar** number of characters.

c. void scset(int row, int col, int nchar, char c);

This function writes character **c** **nchar** number of times starting at coordinates **row** and **col.**

d. void scsav(int row, int col, int nchar, char *v);

This function copies into **v** (which requires 2***nchar** number of bytes storage allocated) **nchar** number of characters (i.e., both the character and its attribute) displayed on the console starting at the coordinates, **row** and **col.**

e. void scbak(int row, int col, int nchar, char *v);

This function writes characters (i.e., both the character and its attribute) from **v** back to the screen at location **row** and **col.**

The speed with which these functions were executed was accomplished primarily by directly addressing the video display memory (instead of depending on DOS or BIOS interrupts). This made them completely unportable and their functions will have to be substituted in the ported software.

J. Video Graphics Functions:

These functions are the basis of all graphics displays used for the current computer tests. The four primitives which draw straight lines are as follows:

a. void ghdraw (int srow, int scol, int erow, int ecol, int color);

This function draws a straight line from starting coordinates, **srow** and **scol,** to ending coordinates, **erow** and **ecol,** in high resolution graphics. The **color** is either 1, for white, or 0, for black.

b. void ghhairs(int lastrow, int lastcol, int newrow, int newcol, int size, int color);

This function moves a cross-hair on the screen by erasing a cross-hair at coordinates **lastrow** and **lastcol** and then drawing a cross-hair at the coordinates, **newrow** and **newcol.** The parameter size is the length in pixels of the vertical hair; the horizontal hair is approximately 2.625 x **size.** The parameter color is 1 for white and 0 for black.

c. void ghdiamd(int lastrow, int lastcol, int newrow, int newcol, int size, int color);

This function is identical to function **ghhairs** except that the figure drawn is a diamond shape instead of a cross-hair.

d. void ghbox(int lastrow, int lastcol, int newrow, int newcol, int size, int color); This function is identical to function **ghhairs** except that the figure drawn is a square box instead of a cross-hair.

The last three functions (i.e., **ghhairs, ghdiamd,** and **ghbox**) could have been done using just the first one, **ghdraw,** because all three figures involved used straight lines. They were divided into separate Assembly language routines for speed, which may not be necessary in the ported software assuming at minimum a 386 processor running at 16 MHz.

There is one more graphics primitive that requires explanation:

void drawpnts(int *pnts, int npts, int row, int col, int color); This function draws individual pixels from array (**npts** by 2) of points, **pnts,** where each pai · of row/column coordinates is biased by the parameters, **row** and **col.** The **color** is 1 for white and 0 for black.

The only purpose of this primitive is to draw a circle. This function would seem to be redundant with other primitives. The reason it was written for the original software was that the time required for the computations for drawing a circle exceeded available time using the slower 8088 CPU. This solution pre-computed the coordinates and stored them in an array which was then repeatedly used to draw and redraw a moving circle target.

The software functions in this group may be readily replaced with runtime library functions from Borland C. However, the functions served by all the video graphic primitives determine critical properties of the tests. Changes will have to be carefully planned in the ported software to achieve the same display properties.

K. Timing Functions:

Four functions are used for measuring elapsed time in the original test software. All use the add-on clock in the Seequa computer which operated independently of the CPU. The four functions are as follows:

a. int timmils(int flg, long *time_seed);

b. int timh(int flg, long *time_seed);

c. int tims(int flg, long *time_seed);

d. float timr(int flg, long *time_seed);

All four functions use a memory location in the calling program to store the absolute time obtained from the add-on clock. The **time_seed** is saved from one call to the next in order to permit computation of elapsed time. On any particular call, the **time_seed** is reset to the current time if **flg** is equal to 0; else it remains unchanged. The **time_seed** must be initialized by one call with **flg** equal to 0 before the first time interval is measured.

The primary difference between the four primitives is the value returned by the function.

The function **timmils** returns the integer value of elapsed time in milliseconds. Because it is an integer (2-byte) function, the maximum interval cannot exceed 32,768 milliseconds (about 0.546 minute).

The function **timh** returns the integer value of elapsed time in 1/100ths of a second. The maximum interval cannot exceed 32,768 hundredths of a second (about 5.5 minutes).

The function **tims** measures the time interval in whole seconds. The function **timr** returns a real (float) value of the time which is accurate to 1/1000th of a second.

The function **timr** might have been the only one needed except that the CPU time for constructing and using the floating point value (without a math coprocessor) was considered too much at the time. This may not be a limitation in porting to faster hardware.

L. **Timed Delay Functions:**

Two primitives were used in the original software to delay execution:

a. void delay(int time_hund);

This function causes a delay in program execution of **time_hund** number of 1/100ths of a second.

b. void mdelay(int time_thous);

This function causes a delay in program execution of **time_thous** number of 1/1000ths of a second.

APPENDIX C

EXAMPLE OF CONVERSION FROM PASCAL TO C CODE

Pascal routine from the original software:

```
function number(const line:lstring;var col,val:integer):boolean;
{ parsing routine: obtain next integer number }
        var  i,j,k,m,mx:integer;
             flag,flag2:boolean;
             t:real;
        begin  {number}
         number:=false;
         flag:=false;
         mx:=ord(line[0]);
         i:=col;
         run_blanks(line,i);
         if line[i]<>'-' then k:=1 else begin k:=-1; i:=i+1 end;
         if i>mx then return;
         t:=0;
         j:=i;
         repeat
          begin
           flag2:=false;
           m:=ord(line[j])-ord('0');
           flag2:=(m>=0) and (m<=9);
           if flag2 then
            begin
             t:=10*t+m;
             flag:=true;
             j:=j+1;
            end;
           end;
         until (not flag2) or (j>mx);
         if line[j]=',' then j:=j+1;
         if (flag) and (t<32768.0) then
          begin
           val:=trunc(t)*k;
           number:=true;
           col:=j
          end;
        end;  {number}
```

Same Pascal function rewritten in C:

```
#include <string.h>
#include <stdlib.h>
```

```
/* declarations */
int number(char *line, int *col, int *val);
void run_blanks(char *line, int *col);

/* function number parses an integer value from the input line read into the character
   string, line, starting at line[col]. If a number is found starting at line[col], then:

   a. the cursor col is incremented to the end of the number,
   b. the numerical value is stored at the address provided by the pointer, val, and
   c. the value of the function is "true" (or 1).

   If no number exists starting at col, then the value of the function is "false", contents
at col and val are unchanged.

   Requires function run_blanks.

   Portability: DOS, UNIX, & OS/2 */

int number(char *line, int *col, int *val)
 {
  int curs,ndigits,sign;
  curs=*col;
  run_blanks(line,&curs);          /* bypass leading blanks on line     */
  if(*(line+curs)=='-')            /* check for minus sign:             */
    {                              /*   set sign to -1 if found and     */
     curs++; sign=-1;              /*   increment cursor.               */
    }
   else
    sign=1;                        /* if no sign then assume '+'              */

  ndigits=strcspn(line+curs,"0123456789\x00");

  if(ndigits)                      /* count number of digits and, if    */
   {                               /* nonzero, then convert the value,  */
    *val=sign*atoi(line+curs);     /* increment the cursor col          */
    *col=curs+ndigits;             /* and return "true."                */
    return(1);
   }
  return(0);                       /* otherwise fail and return "false" */
 }
```

APPENDIX D

EXAMPLE OF CONVERSION FROM ASSEMBLY TO C CODE

Assembly routine from the original software:

```
    title function inkey (Rosse -- 12/26/83)
;
;function inkey(flag:integer):char;
;  if flag=1 then wait for key entry
;    and return it;
;  if flag=0 then return key entry if
;    there was one else return
;    status code of 00h.
;
inky    segment 'prim'
    public inkey
    assume cs:inky
inkey  proc   far
    pop cx
    pop bx
    pop ax
    push bx
    push cx
    cmp al,0
    jnz tag2
    mov ah,6h
    mov dl,0ffh
    int 21h
    jnz tag3
    mov al,0h
    ret
tag2:   mov ah,8h
    int 21h
tag3:   ret
inkey  endp
inky    ends
    end
```

Same function rewritten in C:

```
#include <conio.h>
/*      Function inkey queries keyboard input for next keystroke.
    If no keystroke is in the stream of input characters, then

        a. if flag is "true", it waits for a keystroke and returns its char value.
```

b. if flag is "**false**", then it has the value of 0 x 00.

If keystroke is present, then its **char** value is returned.

Uses C runtime library routines, **kbhit**() and **getch**().

Portability: DOS, UNIX, OS/2 */

```
char inkey(int flag)
if(!kbhit())
if(!flag)return(0);
else while(!kbhit());
return(getch()); }
```